Introduction to Operating Systems

Ben Juurlink   Henk Corporaal

1 Introduction

Building an actual computer is a complex task. In order to cope with the complexity, hardware as well as software designers often think of a computer system as consisting of a hierarchy of **layers** or **levels**. Each level hides certain details from the level below and is therefore more convenient to use.

Tanenbaum [8] distinguishes six levels, which are illustrated in Figure 1. The lowest level, level 0, is the digital logic level or gate level. It is covered in Appendix B of the textbook of Patterson and Hennessy [3] and in the courses Digital Systems I and II (ET1 026 and ET1 027). [Actually, there is another level below level 0, which may be called the realization or VLSI level, but we will not concern ourselves with that level.]

The next level is the microarchitecture level. Here we can see the register file, the ALU, the datapath and control. It was also covered in detail in the textbook.

The next higher level, level 2, defines the instruction set architecture (ISA). It forms the interface between the hardware and software. Two computers are (binary) compatible if they have the same ISA, but the underlying implementation (the microarchitecture) can be different. Indeed, the microarchitecture of the Pentium is very different from the microarchitecture of the 80386, even though they basically implement the same instruction set (the Pentium II also supports a number of instructions specifically targeted to multimedia applications, the so-called MultiMedia eXtensions (MMX)). It is important to realize that the ISA is different from the assembly language level. The latter may contain pseudoinstructions which are not implemented in hardware in order to simplify programming.

The next level, the operating system (OS) level, is the subject of this document. It adds a variety of new facilities to the ISA. In other words, it presents the user with a *virtual machine* which is much easier to use and program than the underlying hardware.

The facilities added by the operating system are called **system calls**. A typical example of a system call is to read a block from a file. A user is not able to perform I/O directly for various reasons. First, it requires exact knowledge of *where* the block is located on *which* device. Second, it requires knowledge of how communication between the processor and the device is established. The OS frees the programmer from all these details through the system call interface. Furthermore, on systems with multiple users the OS must offer some degree of protection so that one user cannot access the files of another user. If users were able to perform I/O directly, it would be much harder if not impossible to provide protection.

Level 4, the assembly language level, is the next higher level. Assembly language programs can also make system calls, which explains why in Figure 1, the assembly language level is placed on top of the OS level. Actually, it is questionable if the assembly language level should be viewed as a separate level, since it does not add any functionality to the OS level, but we follow the classification of Tanenbaum [8]. The available system calls are usually listed in the

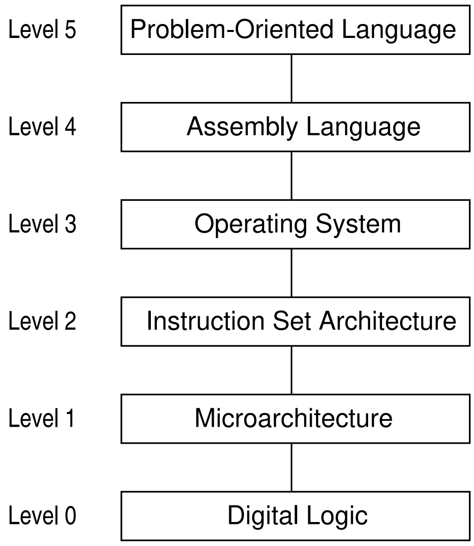

Figure 1: A multilevel computer.

assembly language programmer's manual. For example, Figure A.17 in Appendix A of the textbook lists some of the system calls provided by the SPIM simulator.

The top level is the problem-oriented language level. This is the level seen by application programmers with problems to solve. Depending on the nature of the problem, he or she can select a high-level language (like C, C++, Java, Visual BASIC, etc.) to write a program in. Most languages do not use system calls to perform I/O, but instead provide library functions (like `printf` and `scanf`) which are even more convenient to use than system calls. This problem-oriented language level is the topic of introductory programming courses (like ET1 028) and will also not be covered here.

In order to fully understand why computers are organized the way they are, it is imperative that you have at least a basic understanding of operating systems. Some OS functions require architectural and/or implementation support. For example, systems that support paging usually have a *page table register* which can only be accessed by the OS. As another example, process synchronization (see Section 4) requires instructions that indivisibly read and update a memory location.

This document is organized as follows. In Section 2 we extend the virtual memory model described in Chapter 7. In particular, we describe **segmentation**, a technique that allows different parts of a program to grow and shrink in unpredictable ways. It also allows powerful methods of protection, which is an important issue in systems with multiple users.

Section 3 discusses **concurrent processing**. Most modern operating systems allow many programs to be active at the same time and these programs can belong to different users. An active program, i.e., a program in execution is called a process. The OS must make sure that different processes cannot interfere with each other and that each process gets a fair share of the CPU time. In Section 3 we describe how processes can be created in Linux and how they can communicate with each other. In addition, we sketch how concurrent processing is simulated on a single processor.

When two or more processes can access the same shared data, **synchronization** becomes an important issue. For example, when one process writes data to a file which another process

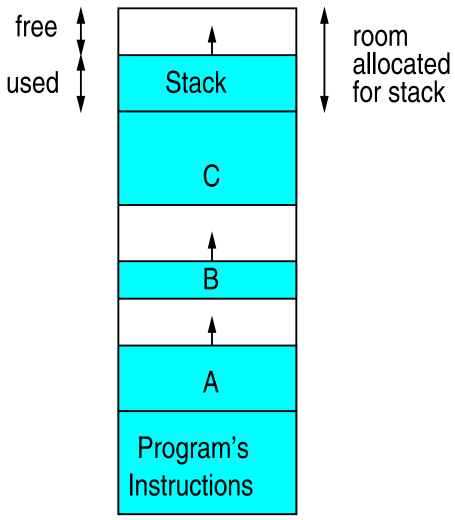

Figure 2: In a one-dimensional address space, one table may run up against another.

should print, it is important that the second process does not start until the first process has written something. Process synchronization is discussed in Section 4.

Finally, in Section 5, we discuss **threads**. Threads are sometimes called "lightweight processes". In Section 5 we get into the nitty-gritty details of how threads are implemented by developing a small threads package for the Intel 80x86 architecture.

2 Virtual Memory

In this section we extend the virtual memory model described in Chapter 7.

2.1 Segmentation

The virtual memory model discussed in Chapter 7 is actually called **paging**. It divides the virtual memory, which is a one-dimensional address space, into fixed-size blocks called pages. These pages can be placed in any page frame in memory and the mapping of virtual addresses to physical addresses is done via a page table.

An important limitation of the paging technique is that it does not support data structures that grow and shrink in unpredictable ways. Consider, for example, an application that uses three tables, A, B and C. These tables grow and shrink but neither the programmer nor the compiler knows how much space is needed for each of them. In a one-dimensional address space like the one described in Chapter 7, the tables must be allocated to contiguous blocks of (virtual) memory. Furthermore, space must also be allocated for the program's instructions and for the stack, which also grows and shrinks unpredictably. Then, no matter where we place them, there is always the chance that one table will run up against another. This situation is illustrated in Figure 2. In this figure, table C has bumped into the room allocated for the stack.

A solution to this problem is called **segmentation**. Each segment is an independent address space and consists of a linear sequence of addresses, from 0 to some maximum. Different segments can have different lengths and, furthermore, the length of segment can change during execution. For example, the length of the stack segment increases when something is pushed onto the stack and decreases when something is popped off.

Different segments can grow or shrink independently. Since each segment constitutes a separate address space, it cannot run up against other address spaces. In the example above, the tables A, B and C, the stack and the program should each be given a separate segment.

It is important to realize that the (assembly) programmer is aware of segmentation. Unlike paging, segmentation is not **transparent** at this level. To specify an address in a segmented memory, the programmer must supply an address consisting of two parts: a segment number and an address within the segment.

Segmentation has other advantages besides being able to support data structures that grow and shrink. It also supports elaborate forms of protection. For example, one segment may contain the program code, and since it is generally considered a bad idea to write self-modifying code, this segment could be specified as execute-only, which means that it is allowed to execute this segment but not to read from or write to this segment. Furthermore, the stack segment could be specified as read/write, which implies that an attempt to branch to it will cause an error. Segmentation also facilitates sharing of programs between several users. If many users simultaneously run an Internet browser like `netscape` or `explorer`, only one copy of the program needs to be in memory, which implies that less memory is required.

2.2 Implementation of Segmentation

Segmentation can be implemented in two ways: **swapping** or **paging**. With swapping, entire segments have to be stored contiguously in physical memory. If a segment is referenced that is currently not in memory, it needs to be brought in entirely. If there is no room for it, one or more segments have to be written back to secondary storage (e.g., hard disk).

Swapping suffers from what is called **external fragmentation**. This means that a segment cannot be placed in memory because there is not a *contiguous* block of memory available in which the segment can be placed, even though the *total* amount of free memory is enough. This situation is illustrated in Figure 3. The total amount of free memory is 7KB, but the largest contiguous free block is 4KB. So, a new segment of 6KB cannot be placed in physical memory, unless one or more segments are written back to disk to make a hole large enough for it to fit, or the segment are *compacted* to create one large hole, as illustrated in Figure 3(b). In either case, precious CPU cycles are wasted.

The other way to implement segmentation is by dividing each segment into fixed-size pages which can be placed anywhere in physical memory. In this scheme, called **paged segmentation**, some pages of a segment can be in physical memory while the other pages are stored on secondary storage. This is very much like ordinary paging as described in Chapter 7 of the textbook, except that each segment now has its own page table. A virtual address consists of two parts: a segment number and a segment address. The segment number is used to index the segment table. Each segment table entry, contains a pointer to the page table of the corresponding segment. In addition, it can contain the access rights (e.g., read, read/write, or execute-only) granted for that segment. The segment address is further subdivided into two parts: the virtual page number and the page offset. As in ordinary paging, the virtual page number is used to index the page table in order to obtain the physical page number, which then is concatenated with the page offset to obtain the physical address. Figure 4 illustrates how address translation in a paged, segmented virtual memory is performed. If you compare this table to Table 7.22 in the textbook, you should notice that they are very

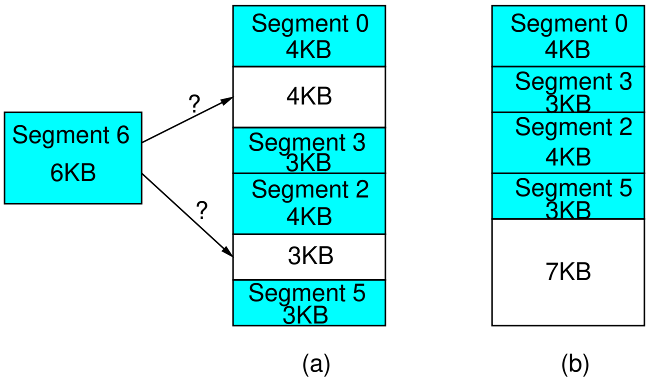

Figure 3: (a) External fragmentation: a new segment larger than 4KB but smaller than 7KB cannot be placed in physical memory, even though the total amount of free memory is sufficient, (b) removal of fragmentation by compaction.

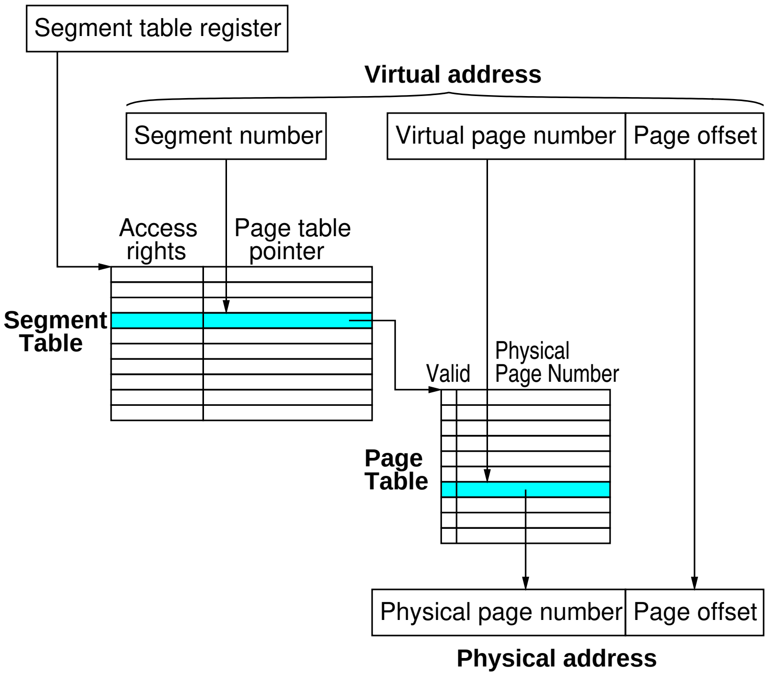

Figure 4: Address translation in a paged, segmented virtual memory.

similar except for the addition of the segment table.

The Pentium II's virtual memory system supports paging, pure segmentation as well as paged segmentation. It also supports elaborate forms of protections. The reader is referred to [8] for an excellent description of the Pentium II's virtual memory system.

2.3 Segments in UNIX

UNIX does not implement segmentation as described above. It does, however, divide the virtual address space into three parts called segments (see Figure 5), but there is only one page table for the entire process and not one for each segment.

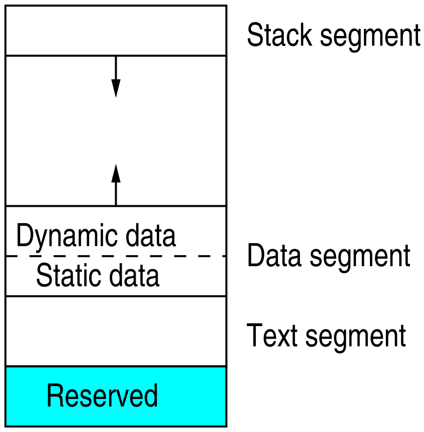

Figure 5: Segments in UNIX.

The first segment starts at a certain address (in the SPIM simulator at address 0x400000) and contains the program's instructions. This segment is called the **text segment**. It is usually execute-only. Below the text segment is a block of memory reserved for the OS.

The next segment, immediately above the text segment, is called the **data segment**. It consist of two parts, one for static data and one for dynamic data. The static data part contains variables whose size is known to the compiler and that exist during the program's entire execution. Global variables are static data.

Immediately above the static data is the dynamic data part. This data is allocated by the program when it executes. In C, we can use the `malloc` library function to allocate a new block of memory. When `malloc` runs out of memory, it can ask the OS to add more pages to the program's virtual address space. In UNIX, this is done using the `sbrk` system call. As indicated in Figure 5, the dynamic data part of the data segment grows from low to high addresses.

The third segment, the **stack segment**, starts at the highest memory address and grows in exactly the opposite direction. In this way, the two dynamically expanding segments are placed as far apart as possible and they can grow until they use up the entire virtual address space. You should now understand why the stack grows from high to low addresses in the MIPS processor.

All modern operating systems allow several programs to be active at the same time. An active program is more correctly called a **process**. How this is implemented is discussed in the next section, but for now it important to understand that each process has its own virtual address space.

3 Concurrent Processing

What was covered in the textbook is enough to completely understand how the instructions of a single program are executed one at a time until it completes. In many situations, however, it is desirable to have multiple programs, or **processes**, running at the same time. A process can be thought of as a program in execution.

Of course, on a system with one CPU, the processes do not truly execute simultaneously. Instead, the processor is time shared among several processes. This means that each process is run for a short while, and then another process is given control of the CPU. We refer to this as **simulated concurrent processing** in order to differentiate it from true concurrent processing, as can be done on systems with more than one CPU. **Time sharing** is another term used to denote simulated concurrent processing.

What are the reasons for supporting simulated parallel processing? First, it provides a way of overlapping I/O with computation. I/O devices are very slow and instead of letting the CPU wait until the running process finishes its I/O (on a page fault, for example), the OS can temporarily suspend the current process and let another process run. In this way, a higher degree of CPU utilization can be achieved. Operating systems that support this are often called **multiprogramming** systems.

Second, in the old days, computers were very expensive and by dividing the CPU between several user processes, the computer could be made more productive. In this way, several users could use the same computer simultaneously. Provided the computer is fast enough (and nowadays they are), each user thinks that he or she is the sole user of the system, since the computer immediately reacts to his or her commands. It helps that interactive programs seldomly require the full processing power of the computer.

Third, some computations can be more conveniently programmed as two or more processes running in parallel than as one process. Consider, for example, a Web browser that downloads a large file such as an audio or video clip. We do not want to wait until the entire file is downloaded before surfing to other pages. Furthermore, maybe we want to play the video clip before it is entirely downloaded. If there were only one process, this process simultaneously has to perform the normal duties of the browser (like reacting to your mouse clicks, send requests to Web servers, display the Web pages, etc.), keep track of the status of the file (for example, check whether the download has finished), *and* play the video clip at the same time. Think about how to program this for a while. What if you download ten files at the same time? You can program this, but then you effectively implement time sharing yourself. It is much more convenient to have multiple processes: one that downloads the file, one that performs the normal duties of the browser, and one that decodes and displays the video clip.

This section is organized as follows. Section 3.1 describes how processes can be created in Linux and how these processes can communicate with each other. After that, in Section 3.2, we sketch how concurrent processing can be simulated on one processor. Finally, in Section 3.3, we discuss the hardware required to enforce process protection.

3.1 An Example: Linux

Linux is a free UNIX-type operating system originally created by a young Finish student named Linus Torvalds. UNIX was developed at Bell Labs (AT&T) in the early seventies. It soon became very popular at universities, not in the least because it was supplied with the complete source code. The commercial success of UNIX, however, was limited, because at a certain point in time, two incompatible versions were widely used: Berkeley UNIX and System V, developed by AT&T. As a result, software vendors could not assume that their programs would run on any UNIX system. Microsoft, on the other hand, guaranteed them that their software would run on any MS-DOS system.

As an aside, when IBM came to Bill Gates because they needed an operating system for their PC, he was selling UNIX licenses. However, the processor that IBM selected for their PC (the Intel 8088) did not have enough memory to run UNIX. IBM, therefore, asked Bill Gates if his company could write an operating system for their PC. He accepted and his company produced the Microsoft Disk Operating System (MS-DOS). It, and its successors Windows 95/98, made Bill Gates the richest man in the world.

Linux is a true multi-tasking system. In fact, when you give a command to the command interpreter (called the *shell*), a new process is started that executes the command. You can either wait for this process to terminate by just giving a command to the shell, or not wait by putting an ampersand (&) behind the command. So, when you enter the command

```
% netscape
```

where % is the prompt, you can no longer give commands to the shell, because it waits for netscape to terminate. But when you enter

```
% netscape&
```

the shell will immediately prompt you for another command and `netscape` will run "in the background". The shell is itself a process that can create (*spawn*) other processes when needed.

Processes can communicate with each other via a structure called a **pipe**. There are other interprocess communication mechanisms in Linux but the pipe mechanism is probably the easiest one. A pipe is a kind of buffer in which one process can write a stream of data and the other process can take it out. For example, if you want to know how many files there are in the current directory, you can enter

```
% ls | wc -w
```

This means that the output of the command `ls`, which lists the files in the current directory, is used as input for the command `wc -w`, which counts the number of words in a file. What actually happens is that two processes are created and both of them are run (pseudo) concurrently. Furthermore, there is communication channel between them where the first one writes to and the second one reads from so that they can cooperate to form a single task.

You can find a lot of information about Linux, including ftp-sites where you can download the source code for free, on `www.linux.org`.

3.1.1 Process Creation

Let us now look at how processes can be created in Linux. A Linux process can create a subprocess by executing the `fork` system call. The original process is called the **parent process** and the new one is called the **child process**. This gives rise to a tree of processes, as illustrated in Figure 6. In fact, every Linux process is a child process of some other process, except for the root process `init`. When you have access to a Linux system, you can use the command `top` to see a list of all processes and the amount of resources (CPU time, memory) they use.

Immediately after the `fork`, the child process is an *exact clone* of the parent process. This means that the text segments and data segments of the two processes are identical. How then, you might wonder, can we infer which process is the child and which is the parent? This is made possible because each process has a unique identifier, called the process identifier (PID). This value is determined by the `getpid` system call, which has the following prototype[1]:

[1]If you are not an experienced C programmer: In C, a function can be declared before it is defined. Such a declaration is called a *function prototype* or just *prototype*. This facilitates separate compilation because the compiler can check if the function is called correctly

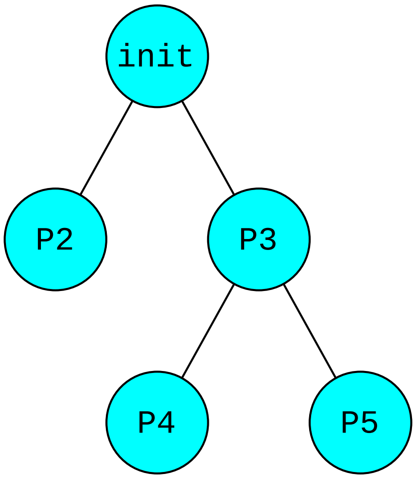

Figure 6: In Linux, the processes form a tree.

```
pid_t getpid(void);
```

(the `pid_t` data type is just a synonym for `int`). Furthermore, the `fork` system call also returns a PID which is equal to the child's PID for the parent and 0 for the child. So, the following program prints the PID of the parent twice and the PID of the child once:

```
#include <unistd.h>
#include <sys/types.h>

main()
{
   pid_t pid_value;

   printf("PID before fork(): %d\n", (int)getpid());

   pid_value = fork();
   if (pid_value==0)
      printf("Hello from child. PID = %d\n", (int)getpid());
   else
      printf("Hello from parent. PID = %d\n", (int)getpid());
}
```

Notice that a C program that uses the `fork` and `getpid` system calls must include the standard header file `<unistd.h>`, and in order to use the `pid_t` type it must include the header file `<sys/types.h>`.

As mentioned in the introduction, the shell is a process that creates a process every time you enter a command. However, immediately after the `fork`, the new process is an exact duplicate of the shell. How then do we get it to execute the command? The answer to this question is the `exec` system call. Its function is to replace the text and data segments of the calling process by some other program. There are many `exec` system calls. In the next example, the `execv` system call is used, which has the following prototype:

```
int execv(char *pathname, char **argv);
```

The pathname is the *full* path of the command to execute, for example, /bin/ls. The argv parameter contains the command-line parameters you might want to pass to the new program.

Normally, the exec system calls do not return. Since their function is to replace the text and data segments of the calling process by some other program, there is simply nothing to return to. However, when they fail, for example because the command specified by pathname does not exist, they will return with an error code. The following program accepts any number of parameter and will try to execute the command specified by the first parameter.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main(int argc, char *argv[])
{
   pid_t pid_value;

   if (argc==1)
   {
      printf("Usage: run <command> [<parameters>]\n");
      exit(1);
   }

   pid_value = fork();
   if (pid_value==0) /* child */
   {
      execv(argv[1], &argv[1]);
      printf("Sorry, couldn't run that\n");
   }
}
```

This program should be compiled and the executable must be stored in the file run. Notice that the first argument of execv is argv[1], not argv[0]. In C, argv[0] is a string corresponding to the name of the program (in this case run). So, argv[1] is the name of the command we want to execute.

The following sequence shows how the run command is used.

```
% run ls -l
Sorry couldn't run that
% run /bin/ls -l
total 1
-rw-r--r--   1 ben      other       64045 May  9 18:30 caos3.tex
```

In the first case, the execv call failed because the full pathname must be given.

In the example above, the parent process (run) does nothing, so it simply terminates. This means, however, that the child process can become an orphan, i.e., a process without a parent. If we do not want this to happen, we must let the parent process wait for the child process to terminate. This can be done using the wait system call, which has the following prototype:

```
pid_t wait(int *status);
```

This call suspends (blocks) the calling process until one of its children terminates, so if a process has more than one child it wants to wait for, it has to execute several `wait` calls. The return value is the PID of the child process that terminated.

The following program shows how the `wait` system call can be used:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

main(int argc, char *argv[])
{
   pid_t pid_value;
   int   status;

   if (argc==1)
   {
      printf("Usage: run <command> [<parameters>]\n");
      exit(1);
   }

   pid_value = fork();
   if (pid_value==0) /* child */
   {
      execv(argv[1], &argv[1]);
      printf("Sorry, couldn't run that\n");
   }
   else /* parent */
      wait(&status);
}
```

3.1.2 Interprocess Communication

We now discuss how interprocess communication (IPC) can be performed in Linux. The simplest IPC mechanism in Linux is the **pipe**. A pipe is a fixed-size buffer (usually 4KB or so), which can be read and written like a file (i.e., sequentially/byte-for-byte), using the `read` and `write` system calls. A lot of the strength of Linux lies in the fact that almost anything can be treated like a file.

In order to create a pipe, the `pipe` system call must be used, which has the following prototype:

```
int pipe(int fd[2]);
```

This call returns -1 on error and two so-called *file descriptors* which must be used in subsequent `read` and `write` system calls. After calling `pipe`, `fd[0]` contains the file descriptor for the read-end of the pipe, while `fd[1]` contains the file descriptor for the write-end. The `read` and `write` system calls have the following prototypes:

```
int read(int fd, void *buf, size_t nbytes);
int write(int fd, void *buf, size_t nbytes);
```

These calls read from respectively write to the pipe associated with the file descriptor `fd`. In case of a `read`, `nbytes` are read from the pipe and stored in the block of memory pointed to by `buf`. In case of a `write`, `nbytes` are written from the memory block pointed to by `buf` to the pipe.

What happens if a process attempts to read from an empty pipe? The answer is that the process is blocked until some other process writes something to the pipe. Similarly, if a process attempts to write to a full pipe, it is also blocked until some other process reads something from the pipe, thereby making some room. This synchronization mechanism is very powerful since it allows two processes, a reader and a writer, to run at different speeds.

There is still one problem that needs to be solved. How do two processes know which file descriptors to use? Since only one of them should create a pipe using the `pipe` system call, how does the other one know the correct file descriptor? The answer to this question lies in the fact that the `fork` mechanism makes it very easy to pass values from the parent to the child. Since the child is a clone of the parent, the child will have access to the correct file descriptors if the parent previously executed a `pipe` system call[2].

The following program prints the famous "Hello world!" message. However, to make things complicated, we use two processes. The parent process writes the message to a pipe and the child process reads the message from the pipe and prints it.

```
#include <stdio.h>
#include <unistd.h>

main()
{
   int  fd[2],
        i,
        status;
   char msg[13] = "Hello world!",
        c;

   if (pipe(fd)==-1)
   {
      printf("Error creating pipe\n");
      exit(1);
   }

   if (fork()) /* parent process is pipe writer */
   {
      close(fd[0]); /* close read-end of pipe */
      for (i=0; i<12; i++)
         write(fd[1], &msg[i], 1);
      wait(&status);
   }
   else /* child process is pipe reader */
   {
      close(fd[1]); /* close write-end of pipe */
      for (i=0; i<12; i++)
      {
```

[2]This is also a weakness of the pipe mechanism. Only processes with a common ancestor can communicate through a pipe.

```
            read(fd[0], &c, 1);
            printf("%c", c);
        }
        printf("\n");
    }
}
```

3.2 Implementation

We now turn to the question: how is concurrent processing simulated on one processor? Do we need special instructions to switch from one process to another? The answer is: no, except that all processor state (such as the contents of the registers, the PC, the page table register etc.) should be readable/writable. In principle, the OS can switch from one process P to another P' by loading the address of an instruction of P' into a register (say `$k0`, one of the two registers reserved for the OS) and by executing the instruction `jr $k0`.

Of course, things are not all that simple. The OS should be able to restart the process P later, so it has to store all information needed to do so. This information is called the **context** of a process and the act of switching from one process to another is called a **context switch**. The context of a process completely specifies the current status of the process.

The OS stores the process context information in a record (structure), called the **process control block** (PCB). There is one PCB for each active process. Obviously, the PCB must contain the address of the instruction executed last when this process was taken away from the CPU. This is equal to the value of the program counter (PC) at the time the process was interrupted. In addition, since other processes will be using the same set of registers, the contents of the registers need to be saved on a context switch. Furthermore, in systems that use paging, the PCB should also contain a copy of the page table register. This is not all information that is kept in the PCB. You are encouraged to think about what other information needs to be saved on a context switch. Usually, the PCB also contains bookkeeping information such as the process ID and how long the process had control of the processor.

If you have access to a Linux system, the `task_struct` structure defined in the file `/usr/src/linux/include/linux/sched.h` contains the PCB. This structure contains a field of type `struct thread_struct` (defined in `/usr/src/linux/include/asm/processor.h`) in which the registers are saved.

Question 1 *Some computer organizations do not write the result of a comparison instruction in a general purpose register (such as the MIPS instruction* `slt $t0, $s3, $s4`*), but write it to a special register called the* condition code register*. Does this register need to be saved on a context switch?*

We now sketch how switching from one process to another can be implemented. In order to ensure that each process gets a fair share of the CPU time, nearly all computers have an electronic timer or clock that periodically generates an interrupt. Now suppose the timer generates an interrupt. As we know from Section 5.6 and Section 7.4 in the textbook, when an interrupt occurs, the address of the interrupted instruction is saved in the exception program counter (`$epc`) register. After that, control is transferred to an OS routine called the **interrupt handler**. The first thing this routine will probably do is to save the contents of the registers in the PCB of the current process, so that it can use the registers. After that, it moves the `$epc` register to a general purpose register (why?), say `$k1`, and stores the contents of `$k1` in the PCB. Finally, if required, it does the same for the page table register and other context information.

Now that the entire context of the current process has been saved, the OS needs to decide which process is allowed to run next. The algorithm used to select the new process is called the **CPU scheduling algorithm**. Once a process has been chosen, the context of that process is restored. This means that the registers and the page table register are restored from the values stored in the PCB. Furthermore, if there is a TLB, the TLB entries need to be cleared (flushed) in order to prevent that the new process uses the TLB values of the old process. Finally, the saved `$epc` in the PCB of the new process is loaded into one of the two registers reserved for the OS, say `$k0`, and control is transferred to the new process by executing the instruction `jr $k0`. Since the OS cannot use any of the other registers to load the saved instruction address in, this explains why `$k0` and `$k1` are reserved for the OS.

The time for a context switch can be large, and during this time no useful work can be performed. Typically, it varies from 1 to 1000 microseconds. A large part of the overhead is caused by the fact that all registers of the current process need to be saved and that all registers of the next process need to be restored. Some systems (such as DECSYSTEM-20), therefore, provide multiple register sets, so that all that needs to be done on a context switch is to change the pointer to the correct register set. However, when there are more processes than register sets, the register contents still have to saved.

Another large part of the overhead is caused by the fact that the TLB is flushed on a context switch, so that the first memory accesses must go through the page table. A common alternative is to add a unique *process identifier* to each virtual address that identifies the current process. This identifier should also be saved in the PCB and is loaded by the OS in a special register on a context switch. The process identifier is concatenated with the tag portion of the TLB, so that a TLB hit occurs only if both the page number and the process identifier match. This technique eliminates the need to flush the TLB on a context switch.

3.2.1 CPU Scheduling

When there is more than one process ready for execution, the operating system must decide which one to run. This part of the operating system is called the *CPU scheduler*.

A good scheduling algorithm must satisfy many criteria. Among the more important ones are:

- **Fairness.** Divide the CPU time among the processes so that each process gets a fair share.
- **Efficiency.** Try to keep the CPU busy 100 percent of the time.
- **Maximize throughput.** This means maximize the number of processes completed in a given time.
- **Minimize response time.** This means minimize the time an interactive user has to wait for a response.

One simple and fair scheduling algorithm is called *round robin*. In this algorithm, each process is given a certain *time quantum* which it is allowed to run. If a process P uses up its time quantum, it is taken away from the CPU and put on the end of a list of ready processes

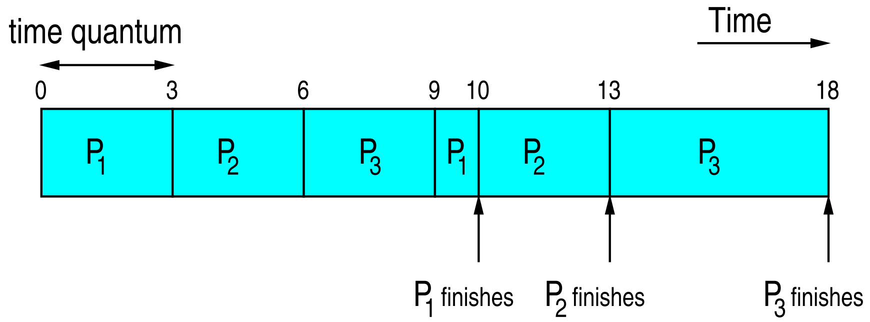

Figure 7: Time diagram that illustrates the behavior of the round robin scheduling algorithm.

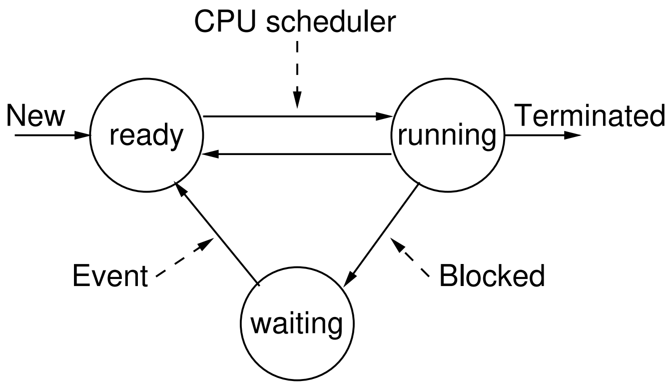

Figure 8: Process state diagram.

(called the *ready queue*). The next process in the queue is allowed to run and P will not be run again until all other processes had a chance to use the CPU. Figure 7 shows how the CPU time is divided between three processes (P_1, P_2 and P_3) that take 4, 6 and 8 time units, respectively. A time quantum of 3 time units is used. Of course, if a process finishes before the end of its time quantum, the CPU will be given immediately to the next process in the ready queue.

In order to improve efficiency and throughput, a context switch is also performed when a process blocks. For example, when a process cannot continue because it generated a page fault, the productivity of the computer can be improved by letting another process run. The OS implements this as follows. Each process is given a state which can be either running (the process is currently running on the CPU), ready (the process is waiting to be assigned to the CPU), or waiting (the process is waiting for some external event, such as I/O completion). The ready processes are linked in a data structure called the **ready queue**, and the waiting process are linked in a **waiting queue**. Furthermore, the CPU scheduler only selects processes from the ready queue. Processes in the waiting queue will remain there until an event occurs that signals that the reason why it was blocked no longer holds (cf. Figure 8).

There are many more scheduling algorithms, but they are beyond the subject of this document. The interested reader is referred to books on operating systems ([9, 7, 5], for example). The file `/usr/src/linux/kernel/sched.c` contains the scheduling algorithm employed by Linux.

3.3 Protection

If several processes belonging to different users can be in memory simultaneously, protection becomes an important issue. In general, a user does not like it if his or her program is modified by a process of a different user. A university professor, for example, does not like it if clever students change their grade while he or she is editing the file containing the grades. Protection was already discussed in Section 2, but the picture is not yet complete.

In order to enforce protection, some of the machine instructions are designated as **privileged instructions**, which can only be executed by the operating system. To support this, the hardware typically contains a bit called the **mode bit**. This bit indicates the current mode of execution: *user mode* or *system mode* (also called kernel mode, privileged mode, or supervisor mode). When the OS is loaded, the hardware starts in system mode. The OS then starts user programs in user mode. Whenever control is given back to the OS (this is called a **trap** to the OS), the hardware switches back to system mode. Thus, privileged instructions can only be executed when in system mode (i.e., by the OS). One has to assume that the OS is free of errors, which is sadly enough not always the case...

As an aside, the processor used in the first IBM PC, the Intel 8088, had no mode bit. As a result, a user program could wipe out the operating system by writing over it with data and multiple programs could write to a device at the same time, with possibly disastrous, or at the very least confusing results. Another example is the following. A typical C programming error is to chase the `NULL` pointer (i.e., to try to load the data pointed to by a `NULL` pointer). Since the `NULL` pointer often points to memory reserved for the OS, the program is almost always terminated with the error message: "memory fault - core dumped" or "segment violation".

Memory protection in systems that uses paging is usually accomplished through the page table. Since every memory access must go through the page table, a process cannot access page frames (in physical memory) that it does not "own"! Because the OS organizes the page tables in such a way that independent virtual pages map to disjoint physical page frames, one process is simply not able to access another one's data. Of course, this also requires that a process is forbidden to change its own page table. Yet, the OS must be able to modify the page tables, for example, when it brings in a page from disk to main memory. Both requirements can be achieved by placing the page tables in the address space of the OS and by making the instruction "load page table register" a privileged instruction.

4 Process Synchronization

In Section 3.1, the Linux pipe mechanism was discussed. Although this is a relatively easy IPC mechanism, in many cases it would be easier if two or more processes could communicate via **shared variables**, i.e., variables that each process could access. For example, if one process produces a value that is needed by ten other processes, the producer could simply write the value to a shared variable which then is read by the other processes.

In Linux, because each process runs in a separate address space, communication via shared variables is not possible straight away. It is possible to make the same block of memory visible in the address spaces of two or more processes, by using the `shmget` system call, but this mechanism is rather awkward. In Section 5 we will discuss threads. Threads are like processes, but they run in the *same* address space. So, when one thread writes some information in memory, it is immediately visible to the other threads.

If two or more processes have access to the same shared variables, access to the shared storage must be carefully synchronized, or otherwise disaster may result. Consider, for example, the computer system of a bank and suppose that two processes, P_c and P_d, are running on this system. The first process handles credits to an account, while the second deals with the debits. Both have access to a shared variable `balance` and P_c executes the statement `balance += amount`, whereas P_d executes the statement `balance -= amount`. Suppose the initial value of `balance` is 500 and that the value of amount is 250. The net effect of this should be, of course, that `balance` remains unchanged. The statements are translated in the following MIPS instruction sequences:

```
/* Process P_c */                      /* Process P_d */
balance += amount                       balance -= amount

lw      $t0,balance                     lw      $t2,balance
lw      $t1,amount                      lw      $t3,amount
add     $t0,$t0,t1                      sub     $t2,$t2,$t3
sw      $t0,balance                     sw      $t2,balance
```

The following might happen. Process P_c runs first and loads the value 500 in `t0`. At that point, a context switch occurs and P_d is allowed to run. It loads the value 500 in `t2` and the value 250 in `t3`, subtracts them and stores the value 250 in the variable `balance`. After that, process P_c is allowed to run again, starting from the place where it left off. It loads the value 250 in `t1`, adds it to `t0` (which contains the value 500), and stores the value 750 in the variable `balance`. Thus, the final value of `balance` is 750 instead of 500. If it were your account you will probably be delighted, but the bank will be less amused. On the other hand, it is not hard to describe a scenario that will yield a final value of 250. In fact, depending on the particular order the instructions are executed, the final value of `balance` can be either 250, 500 or 750.

Situations like this, where the result of a computation depends on the order in which the instructions are executed, are called **race conditions**. The part of the code where the shared memory is accessed with potentially disastrous results is called a **critical section**. In order to prevent race conditions, there must be a way to ensure that not more than one process is in its critical section at the same time. In other words, there must be a way to guarantee **mutual exclusion**; if one process is accessing shared data, then other processes are excluded from doing so too.

In this section, several solutions for the critical section problem are considered. We begin, in Section 4.1 with a very simple solution, namely to turn interrupts off. After that, in Section 4.2, several software solutions (i.e., solutions that require no hardware support) for the critical section problem are discussed. Section 4.3 describes hardware solutions and in Section 4.4 we describe a powerful synchronization mechanism called a semaphore.

4.1 Disabling Interrupts

The simplest way to solve the critical section problem is to let each process disable all interrupts just before entering the critical section and enable them when it exits, as shown in Figure 9. When all interrupts are disabled, no timer interrupt can occur, and therefore no other process can be scheduled and attempt to enter its critical section.

However, it would be very unwise to give the user the power to turn interrupts off. The critical section of the user program could contain a programming error, an infinite loop for

```
...
disableInterrupts();
critical_section();
enableInterrupts();
...
```

Figure 9: Solving the critical section problem by disabling interrupts.

```
while (TRUE)
{
   entry_section();
   critical_section();
   exit_section();
   remainder_section();
}
```

Figure 10: General structure of a process.

example, which would mean that the whole system would “hang”. Furthermore, on systems with more than one CPU, disabling interrupts on one CPU does not stop the processes on the other CPUs. They can still enter the critical section. So, we have to consider alternative solutions.

4.2 Software Solutions for the Critical Section Problem

In this section we look at several software solutions for the critical section problem. Furthermore, we restrict our attention to the case that there are only two processes, P_0 and P_1, both of which have the general structure shown in Figure 10.

4.2.1 Attempt 1 - Strict Alternation

In the first approach, the two processes share an integer variable `turn`, which initially can be either 0 or 1. If `turn==i`, then process P_i is allowed to enter its region. The structure of both processes is shown in Figure 11.

This solution ensures mutual exclusion[3]. At any time, the value of the variable `turn` is either 0 or 1, so only one process will fall through the `while (turn!=i)` loop. The other process will wait in this loop, continuously testing the value of `turn`. This continuous looping is called **busy waiting**.

There are two problems with this solution. First, busy waiting wastes precious CPU time. It should only be used if one knows that the wait time will be short. Second, the solution requires that the two processes strictly alternate in entering their critical sections. If P_1 is much slower than P_0 (for example, because its remainder section takes a long time), P_0 may have to wait even though P_1 does not want to enter its critical section. Suppose that P_0 is in its critical section while P_1 is in its non-critical section. When P_0 leaves its critical section, `turn` is set to 1, and when it quickly finishes its non-critical section and again attempts to

[3]The astute reader may have noticed that a clever compiler might keep the `turn` variable in a register. In that case, no process will enter its critical section after one iteration of the outer loop. We have to assume that shared variables are loaded from memory each time they are used.

```
while (TRUE)
{
   while (turn!=0) /* no-op */ ;
   critical_section();
   turn = 1;
   remainder_section();
}
```

(a) Process P_0

```
while (TRUE)
{
   while (turn!=1) /* no-op */ ;
   critical_section();
   turn = 0;
   remainder_section();
}
```

(b) Process P_1

Figure 11: An attempt to solve the critical section problem.

```
while (TRUE)
{
   flag[0] = TRUE;
   while (flag[1]) /* no-op */ ;
   critical_section();
   flag[0] = FALSE;
   remainder_section();
}
```

(a) Process P_0

```
while (TRUE)
{
   flag[1] = TRUE;
   while (flag[0]) /* no-op */ ;
   critical_section();
   flag[1] = FALSE;
   remainder_section();
}
```

(b) Process P_1

Figure 12: Second attempt to solve the critical section problem.

enter the critical section, it cannot do so even though P_1 is still in its non-critical section. This is undesirable, so another property besides mutual exclusion that a solution to the critical section problem must fulfill is **progress**. This means that no process running in its non-critical section may block other processes that wish to enter their critical section. So, we have to find another solution.

4.2.2 Attempt 2 - Warning Flags

The problem with the first algorithm is that each process does not know if the other process wants to enter its critical section. This problem can be remedied by using a boolean array `flag` of size 2. Both elements of `flag` are initially false, and P_i sets `flag[i]` to `TRUE` in order to warn the other process that it wants to enter its critical section. Figure 12 shows how this algorithm can be programmed in C.

This approach guarantees mutual exclusion. To see this, suppose the contrary, i.e., suppose that both processes are in the critical section at the same time. Then, for P_0 must hold $\texttt{flag[0]} \wedge \overline{\texttt{flag[1]}}$, and for P_1 must hold $\texttt{flag[1]} \wedge \overline{\texttt{flag[0]}}$. This leads to a contradiction, and hence, mutual exclusion is ensured.

However, the described solution is not entirely correct. To illustrate the problem, suppose that P_0 is allowed to run first and sets `flag[0]` to true. After that, a context switch occurs and P_1 sets `flag[1]` to true. Now, P_0 and P_1 will wait in their respective `while` loops forever. This is called a **deadlock**. A correct solution to the critical section problem must also be free of deadlocks, in addition to ensuring mutual exclusion and progress.

```
while (TRUE)
{
   flag[0] = TRUE;
   turn = 0;
   while (turn==0 && flag[1]) ;
   critical_section();
   flag[0] = FALSE;
   remainder_section();
}
```

(a) Process P_0

```
while (TRUE)
{
   flag[1] = TRUE;
   turn = 1;
   while (turn==1 && flag[0]) ;
   critical_section();
   flag[1] = FALSE;
   remainder_section();
}
```

(b) Process P_1

Figure 13: Peterson's solution to the critical section problem.

4.2.3 Attempt 3 - Peterson's Algorithm

By combining the ideas of alternation and warning flags, Peterson [6] devised a correct solution to the critical section problem. His algorithm is shown in Figure 13. The two processes share a boolean array `flag` of size 2 and an integer variable `turn`. Initially, `flag[0]=flag[1]=FALSE` and the value of `turn` is immaterial.

To understand how this solution works, suppose that P_0 sets `flag[0]` to `TRUE` and `turn` to `0`. If process P_1 does not want to enter its critical section, `flag[1]` will be `FALSE`, so the `while` loop test will fail and P_0 will enter the critical section immediately. If P_1 now wants to enter its critical section, it will wait in the `while` loop until `flag[0]` becomes false; an event that only happens when P_0 leaves the critical section. If, on the other hand, P_0 and P_1 want to enter the critical section at about the same time, both will attempt to store their process number in `turn`. Only one of them will succeed, the other store is lost. The process that succeeds in storing its number in the `turn` variable will wait in its `while` loop; the other one will execute its `while` loop 0 times and enter the critical section.

In order to prove that this solution is free of deadlocks, suppose that both processes wait forever in their respective `while` loops. But this cannot be true, since the value of `turn` is either 0 or 1, not both. Finally, to prove the progress property, it suffices to note that when a process P_i is in its non-critical section, it cannot block the other process because it has not set `flag[i]` to `TRUE`. In fact, something stronger can be proven, namely that a process has to wait only once after it has made a request to enter its critical section before that request is granted. To see this, suppose that P_0 and P_1 want to enter the critical section almost simultaneously, and suppose that P_0 is allowed to do so while P_1 busy waits in its `while` loop. When P_0 leaves the critical section, it sets `flag[0]` to `FALSE`, but it may happen that it again wants to enter the critical section before P_1 had a chance to break out of its `while` loop. When P_0 again sets `flag[0]` to `TRUE`, it must also set `turn` to 0. This gives P_1 the chance to enter the critical section. This property of a solution to the critical section problem is called **fairness**.

4.3 Hardware Solutions

We consider one more incorrect solution to the critical section problem. In this approach, a shared boolean variable `lock` is used that controls entry to the critical section. If `lock==0`, the critical section is free; if `lock==1` the critical section is blocked.

```
L:      lw   $t0,lock
        bne  $t0,$zero,L      # if (lock==1) goto L
        addi $t0,$t0,1        # lock = 1
        sw   $t0,lock
        ...                   # critical section
        sw   $zero,lock       # lock = 0
```

This solution is incorrect, because a process could be interrupted just after it has executed the `bne` instruction and just before it has set `lock` to 1. Another process could also test the lock, also find it to be equal to 0 and also enter the critical section. This problem can be solved by disabling interrupts temporarily, as shown in the following assembly code. However, one has to be very careful when to disable and enable interrupts. One cannot simply disable them just before the `bne` instruction and enable them immediately after the `addi` instruction, because then the program would go into an infinite loop if the lock is set.

```
L:      disableInterrupts
        lw   $t0,lock
        beq  $t0,$zero,free   # if (lock==0) goto Free
        enableInterrupts
        j    L                # goto L
Free:   addi $t0,$t0,1        # lock = 1
        sw   $t0,lock
        enableInterrupts
        ...                   # critical section
        sw   $zero,lock       # lock = 0
```

4.3.1 The Test-And-Set-Lock Instruction

The solution above is correct, but – as discussed before – giving the user the power to disable interrupts is not a good idea. Furthermore, on systems with more than one CPU, disabling interrupts on one CPU does not help. Many systems, especially those designed with multiple processors in mind, therefore provide a special instruction to aid with such situations. The most important characteristic of these instructions is that they are **atomic** (i.e., indivisible). An operation is said to be atomic if its execution cannot be interrupted.

One of these special instructions is the Test-And-Set-Lock (`tsl`) instruction. It copies the contents of a memory word into a register *and* writes the value 1 into that memory word. These two actions are executed atomically; if two CPUs execute a `tsl` instruction simultaneously, they will be executed sequentially in an arbitrary order. This is done by *locking* the memory bus until the CPU executing the `tsl` instruction is done. The following assembly code shows how the critical section can be solved using this instruction.

```
L:      tsl  $t0,lock       # $t0 = lock; lock = 1
        bne  $t0,$zero,L    # if ($t0!=0) goto L (lock was set)
        ...                 # critical section
        sw   $zero,lock     # lock = 0
```

Question 2 *Another synchronization instruction is the* `swap` *instruction that atomically swaps the contents of two memory words. Show how it can be used to ensure mutual exclusion.*

```
semaphore mutex = 1;
...
while (TRUE){
   wait(&mutex);
   critical_section();
   signal(&mutex);
   remainder_section();
}
```

Figure 14: The critical section problem solved using semaphores.

4.4 Semaphores

The solutions to the critical section problem presented in the previous section do not easily generalize to more complex synchronization problems. Furthermore, they all require busy waiting. In this section we are going to look at a higher level synchronization mechanism based on the previously discussed hardware or software approaches. In particular, we are going to look at *semaphores*: a synchronization mechanism proposed by Edsgar Dijkstra [2].

A semaphore is an integer variable that, apart from initialization, can only be accessed through two **atomic** operations: `wait` and `signal`. These operations were originally called P (for *passeren*) and V (for *vrijgeven*). The operation `wait(S)` on a semaphore `S` first checks if the value is greater than 0. If it is, it decrements the value by 1 and just continues. Otherwise, i.e., if the value is 0, the process executing the `wait(S)` operation is put to sleep. The `signal(S)` operation increments the value by 1. Furthermore, if one or more processes were sleeping on that semaphore, one of them is woken up and allowed to finish its `wait` operation. Thus, if a `signal` operation is executed on a semaphore with sleeping processes, the value will still be 0 but there will be one fewer process sleeping on it.

4.4.1 Usage

Semaphores can be used to solve the critical section problem as shown in Figure 14. A semaphore `mutex` is used, whose initial value is 1. The first process that executes a `wait` operation on this semaphore will succeed in entering the critical section. Other processes will wait until the first process executes a signal operation. Apart from being easier to comprehend, this solution has the additional advantage that it can be used for any number of processes. The software solutions described in Section 4.2 only work for two processes. It is possible to give a software solution for the general n-process critical section problem (see, e.g., [7]), but it is not easy.

Semaphores can be used for various other synchronization problems as well. For example, if at most k processes can make use of a particular resource at the same time (for example, if there are k printers available), then we can essentially use the same code as in Figure 14, but with the semaphore initialized to the value k. This means that k processes can execute a `wait` operation on this semaphore without being blocked, but any subsequent `wait` operations will cause the calling process to be blocked.

Semaphores can also be used to ensure that certain operations are executed in a particular order. For example, suppose that a process P_1 executes a statement S_1 and that a process P_2 executes a statement S_2, and we want S_1 to occur before S_2. This can be implemented by letting P_1 and P_2 share a semaphore `sync`, initialized to 0, and by letting P_1 execute the statement sequence

```
S_1;
signal(&sync);
```

while P_2 executes the statements

```
wait(&sync);
S_2;
```

Question 3 *Three processes (P_1, P_2 and P_3) cooperate to print the string* `abcabcabcabc....` *P_1 executes an infinite loop in which it prints the character* `a`*, and, similarly, P_2 and P_3 repeatedly print the characters* `b` *and* `c`*, respectively. Give code fragments for all three processes. Do not forget to give the initial value(s) of the semaphore(s). (Hint: your solution should use three semaphores).*

4.4.2 Implementation

The most important thing to remember when implementing semaphores is that it is essential that the `wait` and `signal` operations are atomic. That is, when one process is accessing a semaphore, no other process may simultaneously access the same semaphore, neither through a `wait` nor a `signal` operation. This implies that the `wait` and `signal` operations are themselves critical sections, which can be solved by any of the previously described solutions. The usual way on a single processor system is to implement the `wait` and `signal` operations as system calls, with the operating system briefly disabling interrupts while it is accessing the semaphore. Note that this is different from giving the user the power to turn interrupts off.

On systems with several CPUs, the semaphore operations can be protected using, for example, the `tsl` instruction. This implies that we have not completely eliminated busy waiting. However, busy waiting before a semaphore operation is entirely different from busy waiting to enter a critical section. While the former should take only a few instructions, the critical sections of application programs can take arbitrary long.

The usual way to implement semaphores is to have a list of processes associated with each semaphore. Suppose a process is represented by the following data structure

```
struct process {
   int            state;        /* READY, WAITING, or RUNNING  */
   unsigned       pc;           /* program counter             */
   ...                          /* other information           */
   struct process *next, *prev; /* Next/previous process in    */
                                /* whatever queue              */
}
```

Then, a semaphore can be represented as follows

```
struct semaphore {
   int            value;
   struct process *wq;   /* q of processes waiting on this semaphore */
}
```

Recall that the scheduler only picks processes from the ready queue. So, in order to put a process to sleep, we need to remove it from the ready queue and put it in the queue associated with the semaphore. Similarly, in order to wake up a process, we need to remove it from the waiting queue and put it in back in the ready queue. With this knowledge, the semaphore operations can be implemented using the following pseudo-code.

```
void wait(struct semaphore *s)
{
   disableInterrupts();
   s->value--;
   if (s->value < 0) {
      /* remove the current process from the ready queue */
      /* insert it into s->wq                            */
   }
   enableInterrupts();
}

void signal(struct semaphore *s)
{
   disableInterrupts();
   s->value++;
   if (s->value <= 0) {
      /* remove a process from s->wq    */
      /* insert it into the ready queue */
   }
   enableInterrupts();
}
```

Note that with these definitions, the value of a semaphore can be negative, whereas in the informal definition given at the beginning of this section, they could not. If the value is negative, its magnitude is equal to the number of processes waiting on that semaphore.

One final remark concerns the strategy of inserting and removing processes from the waiting queue. In principle, any strategy can be used, but if we want the semaphore to be **fair** (i.e., each process is given the chance to finish its semaphore operation), then a good strategy to follow is first-come-first-served (FCFS). This can be implemented by inserting and removing processes in a first-in-first-out (FIFO) order.

4.4.3 Two Classical Synchronization Problems

In this final section on semaphores we consider two classical synchronization problems. In the first, two processes, a producer process and a consumer process share a buffer. Semaphores are needed to ensure that (1) the producer does not write into a full buffer, (2) the consumer does not read from an empty buffer, and (3) the producer and consumer do not access the buffer at the same time. The second problem is called the dining philosephers problem and illustrates the danger of creating deadlocks when working with semaphores.

The Producer-Consumer Problem. In the producer-consumer problem (also known as the *bounded buffer* problem), two processes share a fixed-size buffer. The *producer* process produces information and puts it into the buffer, and the *consumer* process takes it out. This is a common situation in computing. For example, a print program may produce characters

```
#define N ...     /* buffer size */

semaphore mutex = 1,
          full = 0,
          empty = N;

void producer(void)
{
   item i;

   while (TRUE){
      produce_item(&i);
      wait(&empty);
      wait(&mutex);
      add_item(&i);
      signal(&mutex);
      signal(&full);
   }
}

void consumer(void)
{
   item i;

   while (TRUE){
      wait(&full);
      wait(&mutex);
      remove_item(&item);
      signal(&mutex);
      signal(&empty);
      consume_item(&item);
   }
}
```

Figure 15: Solution for the producer-consumer problem.

that are consumed by the printer driver (a *device driver* is an OS routine used to hide the differences between the different devices). A pipe also works this way.

Figure 15 shows the solution to this synchronization problem. It uses three semaphores. The semaphore called `full` counts the number of full buffer slots. Since the buffer is initially empty, it must be initialized to 0. The semaphore `empty` counts the number of empty slots. Evidently, it must be initialized to the buffer size `N`. Finally, a semaphore `mutex` is used to ensure that the producer and consumer do not simultaneously add an item to and remove an item from the buffer. This semaphore is needed because otherwise the producer and consumer could access the buffer concurrently and possibly leave it in an inconsistent state.

Question 4 *Suppose the* `wait(&full)` *and the* `wait(&mutex)` *operations were reversed in the consumer process. Would the solution still be correct? Explain.*

The Dining Philosophers Problem. The *dining philosophers problem* is a classical synchronization problem proposed by Dijkstra [2]. The problem is the following. Five philosophers are seated at a round table. All they do is think and eat (as we know, that's all philosophers do...). There are five plates of spaghetti and between each pair of plates is one fork. However, because the spaghetti is very slippery, a philosoper needs two forks to eat, the one from the left of his or her plate and the one from the right. Figure 16 illustrates this situation.

A first attempt to solve this problem is shown in Figure 17. Each fork is represented by a semaphore, whose initial value is 1, meaning that the fork is available on the table. When a philosopher gets hungry, (s)he tries to pick up his or her left and right fork (by performing a `wait()` operation on the corresponding semaphores). If (s)he manages to grab the two forks, (s)he starts eating.

This solution is incorrect because it can lead to a deadlock. To see this, suppose that all

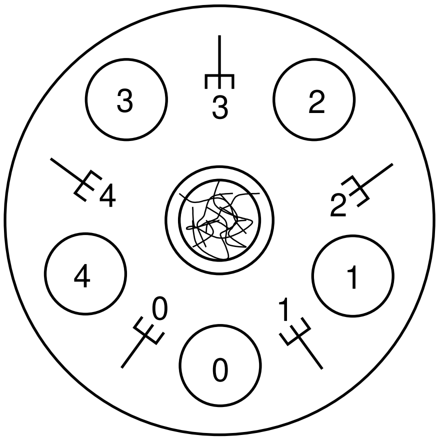

Figure 16: The dining philosophers problem.

```
semaphore fork[5];   /* the 5 forks */
fork[0] = fork[1] = fork[2] = fork[3] = fork[4] = 1;
...
void philosopher(int i) /* i is the number of the philosopher (0..4) */
{
   while (TRUE) {
      think();
      wait(&fork[i]);          /* pick up left fork   */
      wait(&fork[(i+1)%5]);    /* pick up right fork  */
      eat();
      signal(&fork[i]);        /* put down left fork  */
      signal(&fork[(i+1)%5]); /* put down right fork */
   }
}
```

Figure 17: An incorrect solution to the dining philosophers problem.

philosophers pick up their left fork at about the same time. Then, no-one is able to acquire a second fork and start to eat.

A strategy to avoid deadlocks is to impose a numerical order on the resources (in this case, the semaphores), and to require that each process requests resources in an increasing order. This method works because in order for a deadlock to exist, there must be a set of processes $\{P_1, P_2, \ldots, P_n\}$ such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 etc., and P_n is waiting for a resource held by P_1. Figure 18 shows how the previous solution to the dining philosophers problem needs to be modified in order to avoid deadlocks.

This solution does contain a flaw, however. If all philosophers pick up their first fork simultaneously, only one will be able to grab a second fork and start eating. With five forks on the table, it should be possible to allow two philosophers to eat at the same time.

Question 5 *An electrical engineering student proposes the following solution to the dining philosophers problem: all odd-numbered philosophers first pick up their left forks and then their right forks, all even-numbered philosophers first pick up their right forks and then their*

```
void philosopher(int i)
{
   semaphore    *first_fork, *second_fork;

   if (i <= 3) {
      first_fork = &fork[i];
      second_fork = &fork[i+1];
   }
   else { /* i = 4 */
      first_fork = &fork[0];
      second_fork = &fork[4];
   }
   while (TRUE) {
      think();
      wait(first_fork);
      wait(second_fork);
      eat();
      signal(first_fork);
      signal(second_fork);
   }
}
```

Figure 18: Avoiding deadlocks in the dining philosophers problem.

left forks. Does this solution prevent deadlock? Explain. If the solution is correct, how many philosophers can be eating at the same time?

5 Threads

As mentioned before, sometimes having concurrently executing processes that cooperate to solve a certain task is a more convenient programming model than a sequential one in which instructions are performed one after the other. However, processes that do not cooperate have to be protected from each other which implies that interprocess communication is difficult. Furthermore, because of the protection, the time required for a context switch can be large and during this time no useful work can be performed.

The solution to these problems is called **threads**. Threads are like processes, but they run in the same address space. Because of this, they can simply communicate via shared variables. Furthermore, switching from one thread to another can be much cheaper than switching between processes, because they do not need to be protected against each other. The TLB, for example, need not be flushed on a thread context switch. For this reason, threads are also referred to as **lightweight processes**.

Threads are becoming common. The UNIX standard for threads is called **pthreads**. Table 1 lists the most important pthreads functions. Some of these functions require a more detailed description. The function `pthread_join` is used to wait for another thread to terminate. This is sometimes useful because when a process is started there is only one thread: the `main` thread. When this thread creates a new thread and does nothing further (i.e., it simply exits), the newly created thread is also terminated, because threads cannot continue

Function	**Description**
`pthread_create`	creates threads
`pthread_exit`	exits a thread
`pthread_join`	joins with another thread
`pthread_mutex_init`	initialize a new mutex
`pthread_mutex_lock`	lock a mutex
`pthread_mutex_unlock`	release a mutex

Table 1: Basic pthreads functions.

execution if the corresponding process is killed[4]. In order to avoid this, the `pthread_join` function can be used. The synchronization mechanism provided by pthreads is called a **mutex**. Mutexes are like semaphores but they can only take on the value 0 or 1. That is why they are sometimes called binary semaphores. When a thread wants to enter a critical region, it has to lock the mutex (by calling `pthread_mutex_lock`), and when it leaves the critical region, it has to unlock it (by calling `pthread_mutex_unlock`).

The Java language, a programming language heavily used on the World Wide Web, also supports multi-threading. We do not have the time to describe Java threads, but we want to remark that Java uses **monitors** [4] to perform synchronization. A monitor is an even higher level synchronization mechanism than semaphores. A method[5] can be declared as `synchronized`, which means that when one thread is executing this method, no other thread can. So, going back to the implementation of the `wait` and `signal` operations sketched in Section 4.4, in Java it would be easy to protect them by declaring them as synchronized.

Rather than giving a detailed description of pthreads or Java threads, we will develop a small thread package for the Intel 80x86 architecture in this section, so that you will have a full understanding of how threads are implemented. This section is entirely based on Chapter 29 of [1].

5.1 A Tiny Threads Package

We are going to develop a very simple and small thread library that allows threads to be created dynamically at run time. In addition, it supports a communication mechanism very similar to the Linux pipe mechanism. Of course, threads can also communicate via shared variables. We leave it to you to add semaphores to the thread library.

In our thread library, threads are created by calling the `new_thread` function, which has the following prototype:

```
int new_thread(int (*start_addr)(void), int stack_size);
```

The first parameter `start_addr` is a pointer to a function. This function is called when the thread starts executing. In other words, it acts for the newly created thread like the `main` function acts for a process. The `stack_size` parameter specifies how much space, in bytes, should be allocated for the thread's *private* stack. Each thread has its own, private stack

[4]Unfortunately, terms like "terminate" and "kill" are common when talking about threads and processes. Please take no offense at it.

[5]A *method* is very much like a procedure or a function.

because when a thread calls a function, the arguments and the return address are pushed onto the stack, and we do not want the function calls of one thread to interfere with the function calls of other thread. In fact, we will implement a thread context switch by changing the stack pointer. The return value of `new_thread` is 0 if an error occurred (if `malloc` failed) and 1 on success.

The first time `new_thread` is called, usually by the `main` function of a process, the execution of `main` itself is suspended in order to avoid that the process terminates. Every subsequent call to `new_thread` does not suspend the calling thread. Furthermore, only when all the threads created have terminated, control is given back to the `main` function, to the point just after the initial `new_thread` call.

To keep things simple, a thread only relinquishes control of the processor either when it wants to do so or because it cannot continue. This is called **non-preemptive scheduling**. A thread wishing to pass control to another thread can do so by calling the `release` function, which has the following prototype:

```
void release(void);
```

When you use the thread library, you should add calls to `release` anywhere where a thread might loop, so that infinite loops do not prevent other threads from being scheduled.

All threads run in the same address space, so they can communicate via shared (global) variables. However, the thread library does not support semaphores, so a thread which reads from a global variable has no way of knowing that another thread has written something to it. Observe that the mutual exclusion problem is non-existent, since a thread only relinquishes control of the processor when it wants to do.

Instead, the threads library provides a different communication mechanism which is similar to the Linux pipe mechanism. Two threads can communicate by sending/receiving messages to/from a *channel*. If a send occurs before a receive, or vice versa, the sender or receiver is blocked until the other thread arrives at the same point. At this time, the data is passed between the two threads, and the suspended thread is resumed. This concept is called **rendez-vous**, because no-matter which of the two communicating partners (sender or receiver) arrives first at the rendez-vous point, its execution will be suspended until the other partner arrives at the same point.

A channel is created by calling the `get_channel` function, which has the following prototype:

```
int get_channel(int number);
```

The parameter `number` is the number of the communication channel you want to use. If two or more threads wish to communicate via a channel, they have to call `get_channel` with the same argument. The return value is a *channel descriptor* which must be used in subsequent `send` and `receive` calls. If `get_channel` cannot create a new channel (for example, on `malloc` failure), it returns 0.

The first call to `get_channel` on a particular channel number creates a new channel. Subsequent calls on the same channel number simply return the channel descriptor of the existing channel. They do not create a new one.

The `send` and `receive` functions have the following prototypes:

```
int send(int cd, char *buf, int size);
int receive(int cd, char *buf, int size);
```

The `cd` parameter is a channel descriptor returned by a `get_channel` function, `buf` is a pointer to the block of memory from which data will be sent or to which data will be received, and `size` is the size of the data to be transferred (in bytes). If matching sends and receives specify different data sizes, the smaller of the two will be used. The return values of both functions is the actual number of bytes transferred.

5.2 Function Calls

In order to understand the threads library, it is necessary to take a look at how function calls are implemented on the 80x86. This is slightly different from the way it is implemented on the MIPS, because the 80x86 does not have a `jal` instruction that puts the return address in a general-purpose register and jumps to the start of the function. Instead, the 80x86 instruction `call` pushes the return address on the stack and jumps to the start of the function, and the `ret` instruction pops the return address from the stack and jumps to it. In addition, the `ebp` (*extended base pointer*) register is used to find the return address on the stack.

In summary, the following steps are performed when a C-function is called:

- The argument expressions are evaluated and pushed on the stack.
- The `call` instruction pushes the return address on the stack and jumps to the start of the function.
- Inside the function, the value of `ebp` is pushed on the stack, immediately on top of the return address, and the value of the stack pointer, which points to the `ebp` value just pushed, is copied into the `ebp` register, overwriting its previous value, which is now safely stored on the stack.
- Finally, the value of the stack pointer is decremented to make space on the stack for local variables declared inside the function body. Like in the MIPS, the 80x86 stack grows from high addresses to low address.

When a function terminates, the following steps are performed:

- The value of `ebp` is copied into the stack pointer. This automatically discards the local variables and leaves the stack pointer pointing to the saved `ebp` value.
- The top of the stack is popped into the `ebp` register so that the old value of `ebp` is restored. Furthermore, the stack pointer now points to the return address.
- The return instruction `ret` is executed, which pops the return address from the stack and jumps back to the caller.
- Back in the caller the stack pointer is incremented to discard the argument values which were pushed onto the stack. The stack is now in exactly the same state as it was before the function was called.

Figure 19 and Figure 20 show two snap-shots of the stack taken during the execution of the following program.

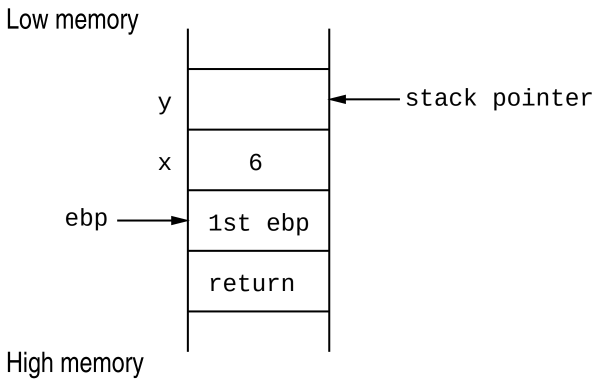

Figure 19: Snap-shot of the stack in `main` just before `twice` is called.

```
main()
{
   int x, y;

   x = 6;
   /* snap-shot one taken here */
   y = twice(x);
}

twice(int n)
{
   int r;

   r = 2*n;
   /* snap-shot two taken here */
   return r;
}
```

The most important thing to remember is that the return instruction `ret` expects the return address on the stack just under the saved `ebp` value.

5.3 Context Switching

In order to be able to switch between threads, we need to save the context of a thread when it is not running. The context of a thread consists of the program counter (called instruction pointer by Intel), the `ebp` register and the stack pointer. The other registers need not be saved, because all general-purpose registers (the 80x86 has only 8) are caller-saved registers. So, since a thread only gives up control of the processor when it calls the `release` function, the thread must save any registers (on its private stack) if it wants to use them after the `release` function has returned.

The thread context is saved in the following structure:

```
struct context
{
   int          ebp;
   char         *stack;
```

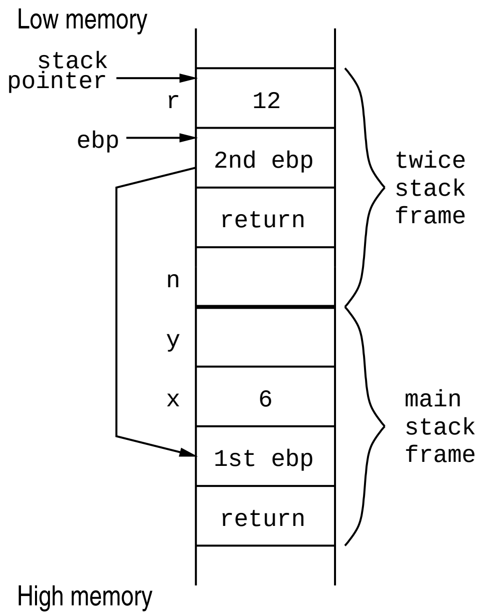

Figure 20: Snap-shot of the stack in `twice` just before `return`.

```
    struct context *next;
    struct context *prev;
};
```

The `ebp` field is used to store the current value of the `ebp` register when the thread is not running. The stack pointer points to an array of bytes which is used as the thread's private stack. When a thread calls a function, the call takes place using the thread's private stack, not the global stack. The `next` and `prev` fields are used to form a circular, doubly linked list of thread contexts. When a thread context switch occurs, the next thread is selected from this list. In addition, we will also maintain a global variable `current` that points to the context of the currently executing thread, and a global integer variable `thread_count` that counts the number of threads.

We are now going to have a look at the `release` function, which is called by a thread when it wants to pass control of the CPU to another thread.

```
int release(void)
{
    /* If there is only one thread, there is no point in doing */
    /* a context switch, so the function returns immediately   */
    if (thread_count<=1)
        return 0;

    /* Otherwise, move current to point to the context of the  */
    /* next thread and call switch context                     */
    current = current->next;
    switch_context(current->prev, current);
    return 1;
}
```

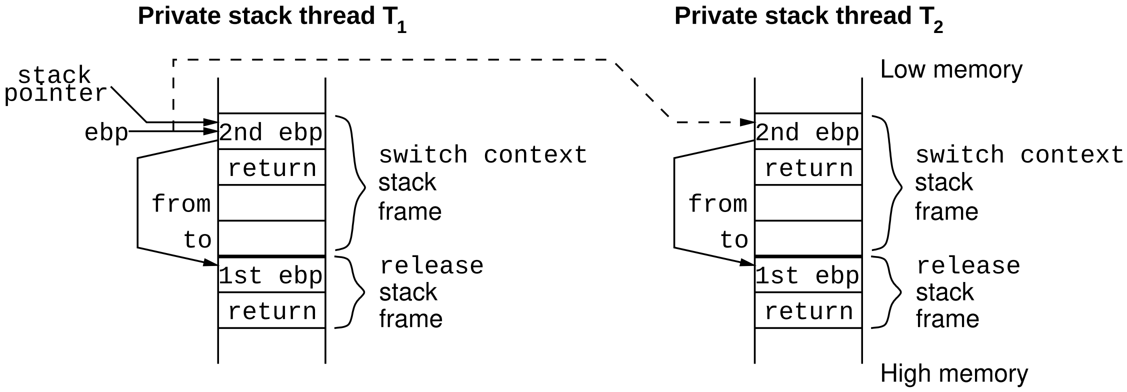

Figure 21: Thread stack layout just before assembly code in `switch_context`. By changing the `ebp` pointer so that it points to the private stack of another thread, a context switch can be implemented. The stack pointer `esp` is changed automatically, on return from the `switch_context` function.

```
static void switch_context(struct context *from, struct context *to)
{
    /* Copy the contents of the ebp register to the ebp field  */
    /* of the from structure and then load the ebp field of    */
    /* the to structure into the ebp register                  */
    __asm__                       /* emit following assembly code   */
    (
        "movl 8(%ebp),%eax\n\t"   /* eax = from                     */
        "movl %ebp,(%eax)\n\t"    /* *eax = *from = from->ebp = ebp */
        "movl 12(%ebp),%eax\n\t"  /* eax = to                       */
        "movl (%eax),%ebp\n\t"    /* ebp = *eax = *to = to->ebp     */
    );
}
```

So, how does this establish a context switch? The "trick" is that both the call to `release` as well as the subsequent call to `switch_context` both take place *using the thread's private stack*. The left hand side of Figure 21 shows the top of the stack of a thread T_1 just before it starts executing the assembly code in `switch_context`. The assembly code saves the contents of the `ebp` register into the `ebp` field of this thread's `struct context`, and then loads the `ebp` register from the `ebp` field of the context of the next thread T_2. If thread T_2 was run previously and released control of the CPU by calling `release`, its private stack will look like in the right hand side of Figure 21. So, after the assembly code, `ebp` points to a place in the private stack of thread T_2. Finally, the `switch_context` function of (now) T_2 returns.

At the end of every function the stack pointer `esp` is set equal to the value in `ebp`, to discard any local variables. The `switch_context` function has no local variables, but the assignment is done anyway. This has the beneficial side effect of moving the stack pointer so that it now points to the correct place in the private stack of the new thread T_2. Finally, the `ebp` register is popped from the current stack (T_2's stack) and the return address is popped from the stack to get back to the calling function. So, the `release` function does not return to the instruction immediately following the `release` call in the code of thread T_1, but it returns to the instruction immediately following a `release` call in the code of thread T_2.

Eventually, all threads will have had a chance to run, so that the thread T_1 gets its next turn to run. When that happens, T_1 resumes execution by an ordinary return from the `release` function. In fact, as far as any of the threads are concerned, they just call `release` from time to time, and when it returns they just continue execution. The threads are unaware of anything unusual happening.

5.4 Creating and Deleting Threads

In order to start a new thread, we have to create a `struct context` for the thread and allocate a block of memory for its private stack. In addition, the thread context has to be inserted into the list of thread contexts. The only tricky part is the initialization of the thread's stack. The `new_thread` function has two parameters: the address of the function where execution of the thread will begin and the size (in bytes) of this thread's private stack.

```
int new_thread(int (*start_addr)(void), int stack_size)
{
    struct context *ptr;
    int esp;

    /* Create a struct context for the new thread */
    /* Return 0 if malloc was unsuccessful       */
    if (!(ptr = (struct context *)malloc(sizeof(struct context))))
        return 0;

    /* Allocate a block of memory for the thread's private stack */
    /* Again, return 0 if malloc was unsuccessful                */
    if (!(ptr->stack = (char *)malloc(stack_size)))
        return 0;

    /* Initialize the stack as shown in Figure 22.                 */
    /* This is the tricky bit, it will be described in detail below. */
    /* Remember: the stack field in the new struct context points   */
    /* to the lowest numbered address where the stack grows from    */
    /* low to high addresses.                                       */
    esp = (int)(ptr->stack + (stack_size-4));
    *(int *)esp = (int)exit_thread;
    *(int *)(esp-4) = (int)start_addr;
    *(int *)(esp-8) = esp-4;
    ptr->ebp = esp-8;

    /* Is this the first thread or not?                           */
    /* If not, insert context in list of thread contexts.         */
    /* If yes, suspend main program and switch to the new thread. */
    if (thread_count++)
    {
        /* There are other threads. Insert the new contexts into  */
        /* the doubly linked list of thread contexts immediately  */
        /* after the context of the current thread.               */
        ptr->next = current->next;
        ptr->prev = current;
        current->next->prev = ptr;
        current->next = ptr;
    }
```

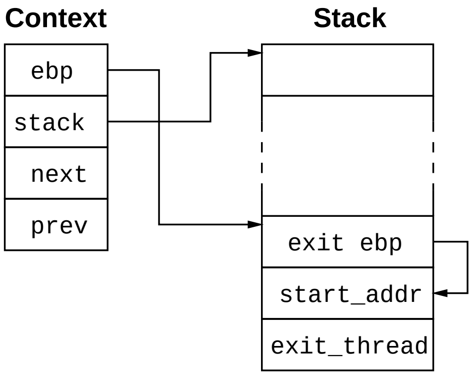

Figure 22: Initial stack layout.

```
    else
    {
        /* This is the first thread. Initialize the list and jump */
        /* to it by calling switch_context(). The context of the  */
        /* main thread is saved in a global variable, so that we  */
        /* can jump back to it later.                             */
        ptr->next = ptr->prev = current = ptr;
        switch_context(&main_thread, current);
    }

    return 1;
}
```

We now explain in more detail how a thread's private stack is initialized. The first time the thread is executed, it has not called `release` yet, so its stack does not look like the right hand side of Figure 21 (there are no `release` and `switch_context` stack frames). How then, you might wonder, can we jump to it? This is done by putting the start address just under the address pointed to by the `ebp` field, as shown in Figure 22. Remember that when `switch_context` returns, it expects the return address just under this address, so when a context switch to this thread occurs, it jumps to the start address of the function where the thread will begin executing.

When a thread's start function terminates, its context must be removed from the list of thread contexts and its stack space must be freed. These actions are performed by the `exit_thread` function. A jump to this function is automatically crafted into the thread's stack, so that you do not need to remember to call this function when the thread's start function terminates. This automatic jump is implemented using a trick similar to the one used to perform a context switch. The code for `exit_thread` is given below:

```
static void
exit_thread(void)
{
    struct context dump, *ptr;

    if (--thread_count)
    {
        /* This thread is not the last. Remove its context from   */
```

```
        /* the list, free the corresponding memory space, and     */
        /* switch to the next thread using a dummy struct context */
        /* to witch from.                                         */
        ptr = current;
        current->prev->next = current->next;
        current->next->prev = current->prev;
        current = current->next;
        free(ptr->stack);
        free(ptr);
        switch_context(&dump, current);
    }
    else
    {
        /* This thread is the last. Free the memory space and     */
        /* switch back to the main program.                       */
        free(current->stack);
        free(current);
        switch_context(&dump, &main_thread);
    }
}
```

5.5 Communication

Finally, we are going to implement the rendez-vous communication mechanism. Two new data structures are used. The first (`struct channel`) is used to represent a channel; there is a `struct channel` for each channel created. The second (`struct message`) is used to represent a message that is blocked, because the communication partner has not arrived yet at the rendez-vous point. Thus, there is a `struct message` for each blocked message. These two structures, along with the declaration of `struct context` and some global variables, etc., are all supplied in a header file `def.h` which should be included in a program that uses the thread library. The full contents of `def.h` is given below.

```
#include <string.h>

struct context                     /* One structure for each thread */
{
    int ebp;           /* Base pointer (stack frame pointer) store */
    char *stack;       /* Pointer to memory block for thread stack */
    struct context *next;      /* Round robin circular list pointer */
    struct context *prev;      /* Round robin circular list pointer */
};

struct channel      /* One structure for each communication channel */
{
    int number;                                    /* Channel number */
    int sr_flag;     /* 0=no queue, 1=send queued, 2=receive queued */
    struct channel *link;           /* Link to next channel in list */
    struct message *message_list;          /* Head of message queue */
    struct message *message_tail;          /* Tail of message queue */
};

struct message       /* One structure for each pending send/receive */
```

```
{
    int size;                           /* Size of message in bytes */
    char *addr;                      /* Pointer to start of message */
    struct message *link;          /* Link to next message in queue */
    struct context *thread;   /* Which thread blocks on this struct */
};

static struct context main_thread;    /* Storage for main() details */
static struct context *current;       /* Currently executing thread */
static int thread_count = 0;       /* Number of threads to schedule */
static struct channel *channel_list = 0;    /* List of all channels */

static int switch_context(struct context *, struct context *);
static int exit_thread(void);
static int rendezvous(struct channel *, char *, int, int);
```

When a new channel is created, a new `struct channel` is created and added to a list of all channels. Whenever a `send` or `receive` is performed on a channel and the matching `receive` or `send` has not yet been issued, a `struct message` is allocated and inserted into the message list associated with that channel. Furthermore, whenever a thread is blocked because the communication partner has not arrived at the rendez-vous point yet, its context is removed from the list of contexts and linked to the message structure. This ensures that a thread cannot be run again until a matching `send` or `receive` is performed on the same channel. Figure 23 illustrates how the various structures are interconnected when there are `send`s and `receive`s pending.

The code for the `get_channel` function is given below. Remember that the first time this function is called with a specific channel number, a new channel is created. The second time it is called with the same channel number, the existing channel descriptor is simply returned.

```
int get_channel(int number)
{
    struct channel *ptr;

    /* Search the list of channels to see if the specified channel  */
    /* number already exists. If so, return the channel descriptor  */
    /* which is simply the address of the associated struct channel */
    /* cast to an integer.                                          */
    for (ptr = channel_list; ptr; ptr = ptr->link)
        if (ptr->number==number)
            return((int)ptr);

    /* Allocate a new struct channel. Return 0 on failure.          */
    if (!(ptr = (struct channel *)malloc(sizeof(struct channel))))
        return 0;

    /* Initialize the fields of the new struct channel. Cast the    */
    /* pointer to this new structure to an int and return it as the */
    /* new channel descriptor.                                      */
    ptr->number = number;
    ptr->message_list = NULL;
    ptr->message_tail = NULL;
    ptr->sr_flag = 0;
```

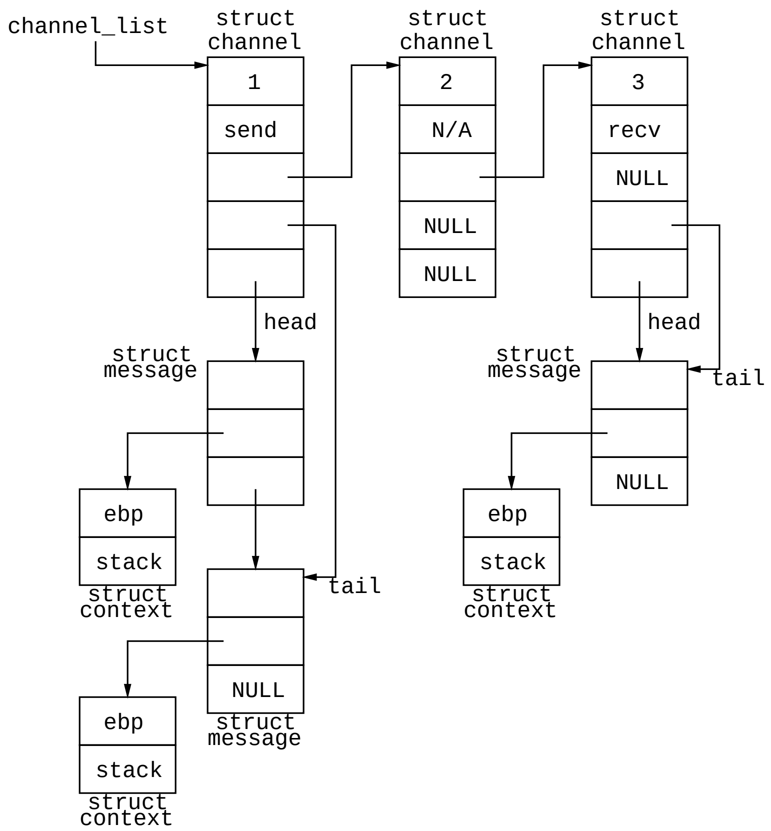

Figure 23: The way channel, message and context structures are interconnected when there are sends and receives pending. For clarity, some of the fields of `struct context` and `struct message` have been omitted.

```
    ptr->link = channel_list;
    channel_list = ptr;
    return((int)ptr);
}
```

The last two functions to be implemented are the `send` and `receive` functions. Because these functions are fully symmetrical, they both make a call to a `static` function `rendezvous` with just an extra flag parameter that specifies the direction of the data transfer:

```
int send(int cd, char *addr, int size)
{
    return(rendezvous((struct channel *)cd, addr, size, 1));
}

int receive(int cd, char *addr, int size)
{
    return(rendezvous((struct channel *)cd, addr, size, 2));
}
```

In both cases, the channel descriptor is cast back to a pointer to a `struct channel`. The added fourth parameter `dir` specifies the direction of the data transfer, where 1 means send and 2 means receive. Because the channel structure has a flag field **sr_flag**, which is 1 if there are sends queued and 2 if there are receives queued, the test to see if there are operations of the opposite type queued is very simple. Furthermore, the return value of both functions is the number of bytes actually transferred.

Finally, the code for the **rendezvous** function:

```
static int rendezvous(struct channel *chan, char *addr,
                      int size, int sr_flag)
{
    struct message *ptr;
    int nbytes;

    /* Check if there is already a communication partner of the      */
    /* opposite type queued. If there is, copy the data and reinsert */
    /* the context of the partner in the list of thread contexts.    */
    /* If not, allocate a new struct message and remove the context  */
    /* of the current thread from the list of threads contexts.      */
    if (sr_flag == 3-chan->sr_flag)
    {
        /* Reinsert the context of the previously blocked thread     */
        /* into the list of thread contexts.                          */
        ptr = chan->message_list;
        chan->message_list = ptr->link;
        ptr->thread->next = current->next;
        ptr->thread->prev = current;
        current->next->prev = ptr->thread;
        current->next = ptr->thread;
        ++thread_count;

        /* Calculate the actual number of bytes to be transferred.   */
        /* Save this number in the size field of the struct message  */
        /* pointed to by ptr, so that the blocked thread can access  */
        /* it on its next turn to run (see also below).              */
        nbytes = (size<ptr->size) ? size:ptr->size;
        ptr->size = nbytes;

        /* Copy the data                                              */
        if (sr_flag==1)
            memcpy(ptr->addr, addr, nbytes);
        else
            memcpy(addr, ptr->addr, nbytes);

        /* Reset the send/receive flag if this message was the last  */
        /* in the list of messages. Return the number of bytes       */
        /* transferred.                                               */
        if (!chan->message_list)
            chan->sr_flag = 0;

        return(nbytes);
    }
    else
    {
        /* There is no communication partner of the opposite type.   */
```

```
        /* Allocate a new message structure and insert it at the end */
        /* of the message queue. Remove the context of the calling   */
        /* thread from the list of thread contexts and link it under */
        /* the message structure.                                    */
        ptr = (struct message *)malloc(sizeof(struct message));

        if (!chan->message_list)
            chan->message_list = ptr;
        else
            chan->message_tail->link = ptr;

        chan->message_tail = ptr;
        ptr->link = 0;
        ptr->size = size;
        ptr->addr = addr;
        chan->sr_flag = sr_flag;
        ptr->thread = current;
        current->prev->next = current->next;
        current->next->prev = current->prev;

        /* Decrement the number of runnable threads and switch to    */
        /* the next thread. Normally, there should always be other   */
        /* threads to execute, but if there aren't (due to a         */
        /* programming error, e.g., if all threads executed a send), */
        /* control is given back to the main thread.                 */
        if (--thread_count)
        {
            current = current->next;
            switch_context(ptr->thread, current);
        }
        else
            switch_context(ptr->thread, &main_thread);

        /* The next time the suspended thread is executed (after the */
        /* communication partner came to the rendez-vous point), it  */
        /* will resume execution here, where the number of bytes     */
        /* transferred can be recovered and returned.                */
        nbytes = ptr->size;
        free(ptr);
        return(nbytes);
    }
}
```

This concludes the description of the threads library.

5.6 An Example: The Sieve of Eratosthenes

One could check whether some number greater than 1 is prime by trying to divide it by other numbers. If this number is prime, its only exact divisors will be itself and unity. A much faster way to check a bunch of positive numbers is by sifting out multiples of the primes. This technique is called the sieve of Eratosthenes (276 BC - 197 BC). This leads to the following algorithm for computing all prime numbers.

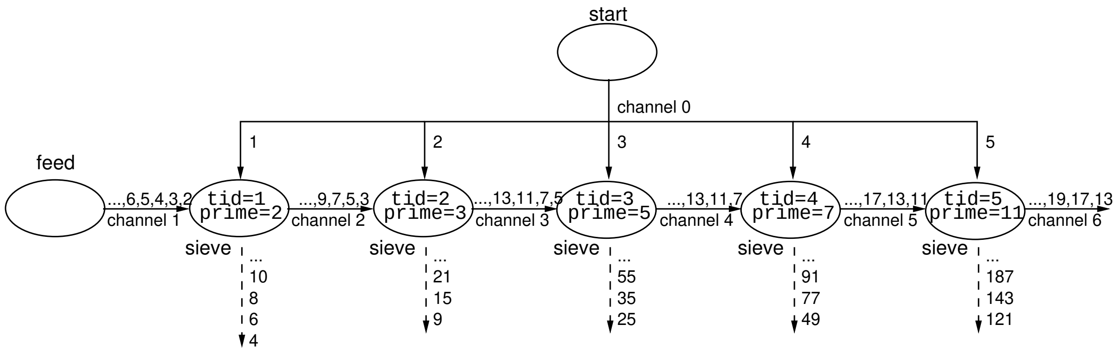

Figure 24: The way threads are organized in the program that computes prime numbers using the sieve of Eratosthenes. Every sieve thread filters out multiples of its prime number (i.e., the first number in the sequence).

We organize the threads in a pipeline. The first thread is responsible for sifting out all numbers divisible by the first prime number (2), the second thread is responsible for sifting out all numbers divisible by the second prime number (3), and so on. All that each thread does is receive numbers from the previous thread and test if they are divisible by its prime p. If they are, it prints a message that the received number is composite. Otherwise, it sends the number to the next thread, which is responsible for the smallest prime number larger than p.

Somehow, a thread must know the channel numbers it should use for input and output. In the program given below, the `start` thread sends each thread a thread ID *tid* via a global channel with channel number 0. After a thread has received its ID, it created two channels numbered *tid* and $tid + 1$. It uses the first channel for input and the second for output. The program also uses an extra thread `feed` that feeds the pipeline with the numbers $2, 3, 4$, etc., up to some maximum number. Figure 24 illustrates the idea. Notice that the first number that each thread receives is the prime number it is responsible for.

```
#define MAX_POT_PRIME       200  /* maximum number tested for primality */

int start(void);
int feed(void);
int sieve(void);

int ch_desc;                           /* global channel descriptor      */

main(void)
{
   ch_desc = get_channel(0);
   new_thread(start, 1024);
}

int start(void)
{
   int i;

   new_thread(feed, 1024);
   new_thread(sieve, 1024);
```

```
   for (i=1; ; i++)
      send(ch_desc, (char *)&i, sizeof(int));
}

int feed(void)
{
   int i, cd_out;

   cd_out = get_channel(1);
   for (i = 2; i<=MAX_POT_PRIME; i++)
      send(cd_out, (char *)&i, sizeof(int));
}

int sieve(void)
{
   int mytid,               /* my thread ID */
       myprime,
       cd_in, cd_out,
       n;

   receive(ch_desc, (char *)&mytid, sizeof(int));
   cd_in = get_channel(mytid);
   cd_out = get_channel(mytid+1);
   receive(cd_in, (char *)&myprime, sizeof(int));
   printf("%dth prime number is %d\n", mytid, myprime);

   new_thread(sieve, 1024);

   while (1)
   {
      receive(cd_in, (char *)&n, sizeof(int));
      if (n%myprime == 0)
         printf("%d is composite\n", n);
      else
         send(cd_out, (char *)&n, sizeof(int));
   }
}
```

We leave it to you to figure out how things exactly work. Notice, however, that threads are created dynamically when needed, i.e., each time a new prime number is found.

Question 6 *You will have noticed that the* `start` *thread and the* `sieve` *threads are infinite threads, but when you execute this program, it will terminate. With your knowledge of the implementation of the threads library, explain why.*

References

[1] Phil Cornes. *The Linux A-Z.* Prentice Hall, 1997.

[2] E.W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages.* 1968.

[3] John L. Hennessy and David A. Patterson. *Computer Organization and Design : the Hardware/Software Interface.* Morgan Kaufmann, 2nd edition, 1998.

[4] C.A.R. Hoare. Monitors, An Operating System Structuring Concept. *Communications of the ACM*, 17:549–557, 1974. Erratum in *Communications of the ACM*, vol. 18, p. 95, 1975.

[5] Gary J. Nutt. *Operating Systems : a Modern Perspective.* Addison-Wesley, 2nd edition, 2000.

[6] G.L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[7] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts.* Addison-Wesley, 5th edition, 1998.

[8] Andrew S. Tanenbaum. *Structured Computer Orgnaization.* Prentice Hall, 1999.

[9] Andrew S. Tanenbaum and A.W. Woodhull. *Operating Systems: Design and Implementation.* Prentice Hall, 2nd edition, 1997.